Advanced Debugging and Performance Analysis

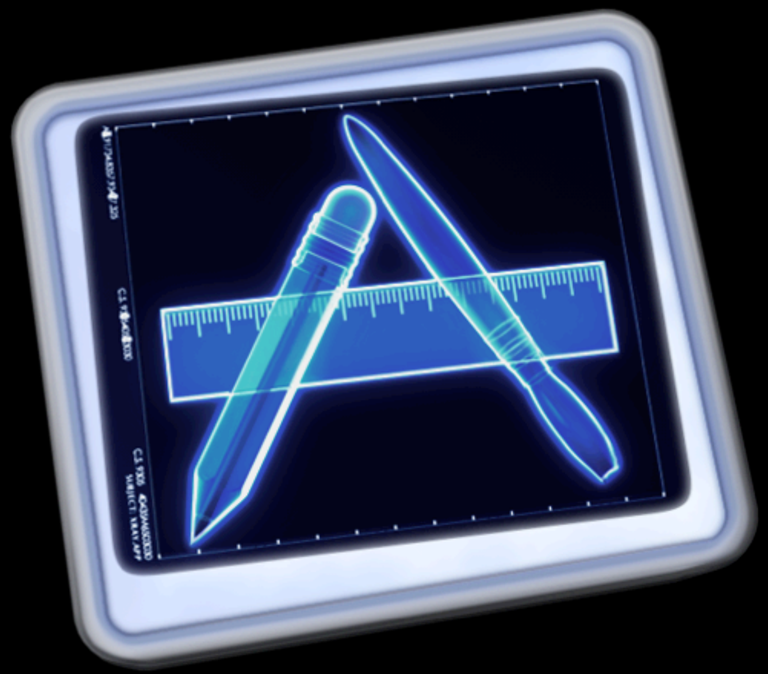
Session 412

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

Performance Analysis

Chad Woolf
Performance Tools Engineer

Today's Session



Performance Analysis

DTrace Instruments

Debugging

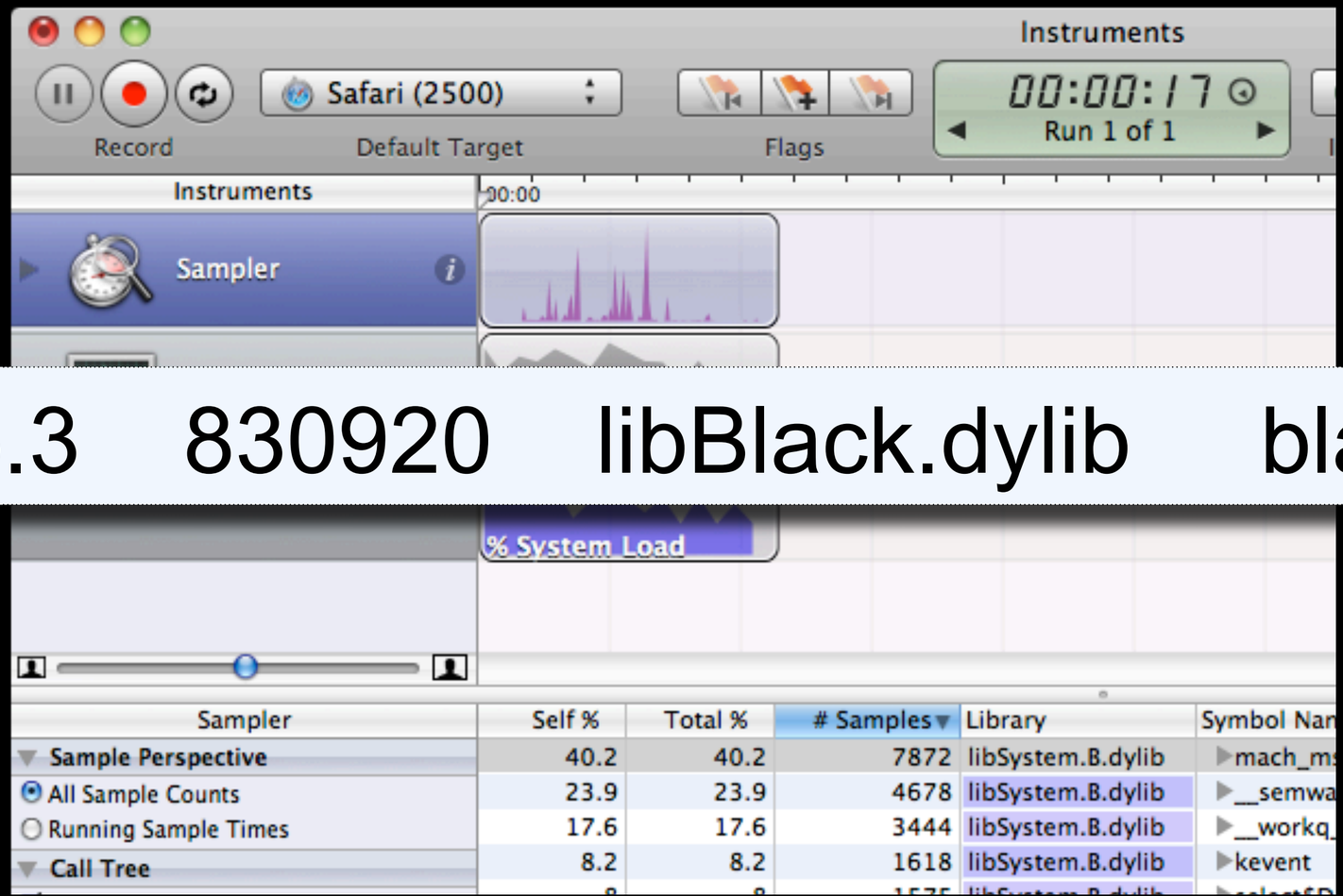
Xcode GDB

First Half—Performance Analysis

- What is DTrace?
- Learn DTrace by example
- Let’s see how it works
- Custom instruments

Where to Start?

Sampling gets us close…



75.3 75.3 830920 libBlack.dylib black_box()

Your Code

Calling black_box()

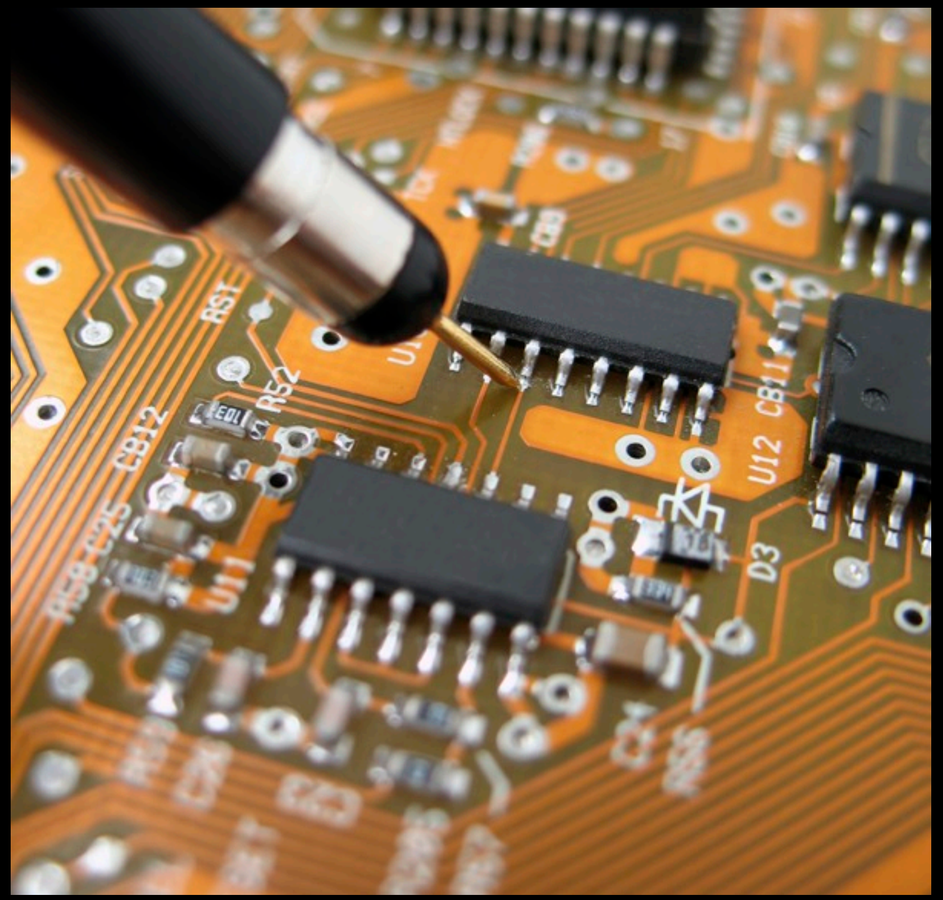
```
...
/* Call black box */
log_start(key);
int rc = black_box(key);
log_end(rc);
...
```

DTrace

DTrace

Like a hardware logic analyzer…for code

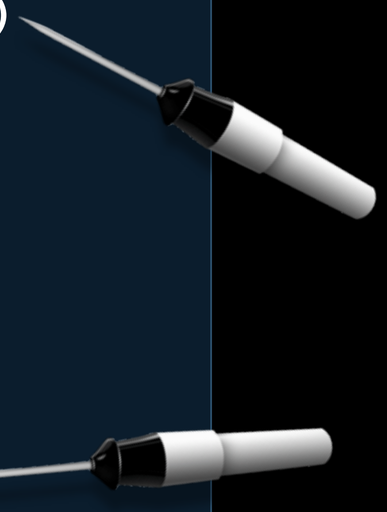
```
...
/* Call black box */
log_start(key);
int rc = black_box(key);
log_end(rc);
...
```



DTrace

Placing probes

```
int black_box(void *key)
{
   ...
   ...
return -1;
  ...
  ...
  ...
  return 0;
}
```



- Symbol table
- dSYM file

DTrace

Placing probes

```
<+0000>  push   %ebp
<+0001>  mov    %esp,%ebp
<+0003>  push   %edi
<+0004>  push   %esi
<+0005>  push   %ebx
<+0006>  sub    $0xcc,%esp
<+0012>  mov    0x18ba4,%esi
<+0018>  mov    0x186d8,%ebx
<+0024>  mov    %ebx,0x4(%esp)
<+0028>  mov    %esi,(%esp)
<+0031>  call   0x11a5c <dyld_stub_objc_msgSend>
<+0036>  mov    0x186d4,%edi
<+0042>  movl   $0x1770c,0x8(%esp)
<+0050>  mov    %edi,0x4(%esp)
<+0054>  mov    %eax,(%esp)
```

- Symbol table
- dSYM file

DTrace Scripts

- Composed of one or more probe clauses

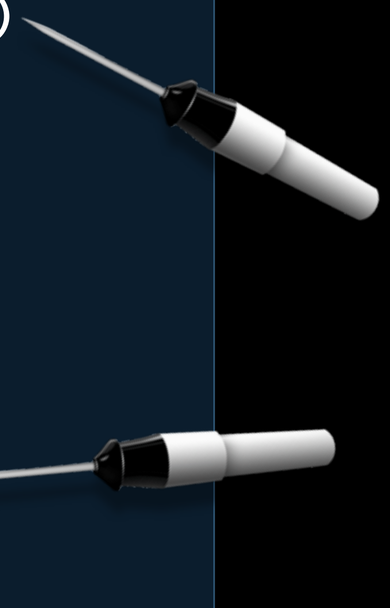
```
probe descriptor
/ predicate /
{
   action statements
}
```

- Written in D language
  - Log data
    - printf(...)
    - ustack(), stack()
  - Record data
    - Thread-safe data accumulators called aggregates
    - Associative arrays

DTrace

Firing probes

```
int black_box(void *key)
{
   ...
   ...
return -1;
  ...
  ...
  ...
  return 0;
}
```



```
pid123:libBlack:black_box:entry
{
    self->entry_time = timestamp;
    self->entry_cpu_time = vtimestamp;
}
```

```
pid123:libBlack:black_box:return
{
    this->ret_offset = arg0;
    this->ret_value = arg1;
    ...
    this->duration =
       timestamp - self->entry_time;
    @avg_duration = avg(duration);
}
```

Probes Are Like Breakpoints

- Suspend the thread
- DTrace runs all actions
- Thread resumes
- DTrace and debuggers don’t get along

Probe Descriptor

```
provider : module : function : name
/ predicate /
{
   action statements
}
```

- Describes events of interest
- Supports wild cards
- It’s the only required part of a probe clause
- List available probes via
  - dtrace -l
  - dtrace -l -n ‘syscall::*stat*:’

Probe Descriptor—Examples

```
syscall : : :
/ predicate /
{
   action statements
}
```

```
pid1017 : libSystem : printf : return
/ predicate /
{
   action statements
}
```

```
objc1017 : : -draw* : entry
/ predicate /
{
   action statements
}
```

Systemwide Providers

DTrace Name	Name in Instrument Builder	Purpose
syscall	System Call	System call entry and return
fbt	Kernel Function Boundaries	Function entry and return in the kernel
proc	Kernel Process	Process and thread creation/termination
plockstat	User-Level Synchronization	pthread lock behavior in user space
lockstat	(Custom)	Lock behavior in kernel
vminfo	(Custom)	Virtual memory activity
io	I/O	Disk I/O
mach_trap	Mach	Mach trap entry and return

http://wikis.sun.com/display/DTrace/Providers

Meta Providers

- Not statically defined, but still there…
- Provider name is composed

DTrace Name	Name in Instrument Builder	Purpose
pid<pid>	User Process	Function call entry and return
objc<pid>	Objective-C	Objective-C method entry and return
profile/tick-<time>	Profile/Tick	Time-based probes

objc1017

tick-100ms

pid1017

tick-1s

profile-10ms

DTrace

Exploring black box

```
int black_box(void *key)
{
   ...
   ...
  return -1;
   ...
   ...
   ...
   return 0;
}
```

myscript.d

```
pid123:libBlack:black_box:entry {

}
```

```
dtrace: 53686 drops on CPU 3

# dtrace -s myscript.d
CPU  id   Probe
3  123   black_box(void *):entry
3  123   black_box(void *):entry
3  123   black_box(void *):entry
3  123   black_box(void *):entry
3  123   black_box(void *):entry
3  123   black_box(void *):entry
3  123   black_box(void *):entry
3  123   black_box(void *):entry
```

DTrace

Exploring black box

```
int black_box(void *key)
{
   ...
   ...
return -1;
 ...
 ...
 ...
 return 0;
}
```

```
pid123:libBlack:black_box:entry {
  @hits = count();
}
```

```
> dtrace -s myscript.d
^C
580209
```

DTrace

Exploring black box

```
int black_box(void *key)
{
   ...
   ...
return -1;
  ...
  ...
  ...
  return 0;
}
```

```
pid123:libBlack:black_box:entry {
  @hits = count();
  @keys = quantize(arg0);
}
```

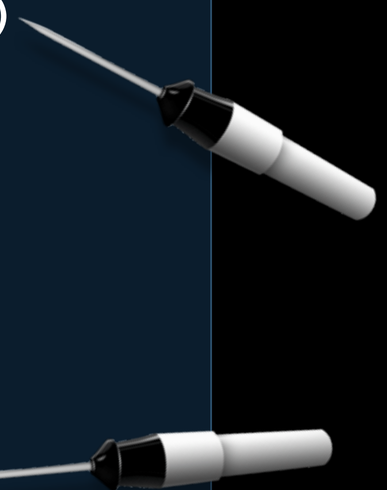
DTrace Output

```
# dtrace -s myscript.d
^C
466535
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@                  380567
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |                                         0
      2147483648 |@@@@@@                                   40387
      4294967296 |@@@@                                     35765
      8589934592 |@                                        9816
     17179869184 |                                         0
```

DTrace

Exploring black box

```
int black_box(void *key)
{
   ...
   ...
return -1;
 ...
 ...
 ...
 return 0;
}
```



```
pid123:libBlack:black_box:entry {
  @hits = count();
  @keys = quantize(arg0);
}

pid123:libBlack:black_box:return {
   @rc[arg1] = count();
}
```

DTrace Output

Quantize

```
# dtrace -s myscript.d
^C
466535

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 380567
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |                                         0
              64 |                                         0
      2147483648 |@@@@@@                                   40387
      4294967296 |@@@@                                     35765
      8589934592 |@                                        9816
     17179869184 |                                         0


  0                 85968
 -1                380567
```

0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 380567

-1 380567

DTrace

Call duration

self-> thread scope
this-> local scope

```
int black_box(void *key)
{
   ...
   ...
return -1;
 ...
 ...
 ...
 return 0;
}
```

```
pid123:libBlack:black_box:entry {
   self->entry_time = timestamp;
}
```

```
pid123:libBlack:black_box:return {
   this->duration =
      timestamp - self->entry_time;

   @duration[arg1] =
       quantize(this->duration);
}
```

DTrace Output

Duration by return code

```
# dtrace -s myscript.d
^C
  -1
           value  ------------- Distribution ------------- count
           35088|@@@@@@@@@@                                 18498
           43009|@@@@@@@@@@@@@@@@@@@@@@                     38058
           51003|@@                                         6030
  0
         value  ------------- Distribution ------------- count
           18240|@@@@@@@@@@                                 11498
           20983|@@@@@@@@@@@@@@@@@@@@@@                     22058
           23087|@@@@@@@@@@@@@@@@                         17030
```

DTrace Example—black_box()

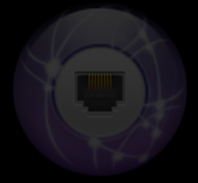
1. black_box() was being called a lot
2. It was mostly called with zeroes
3. Zeroes caused it to fail
4. Failing is slow

Demo

Having fun with DTrace

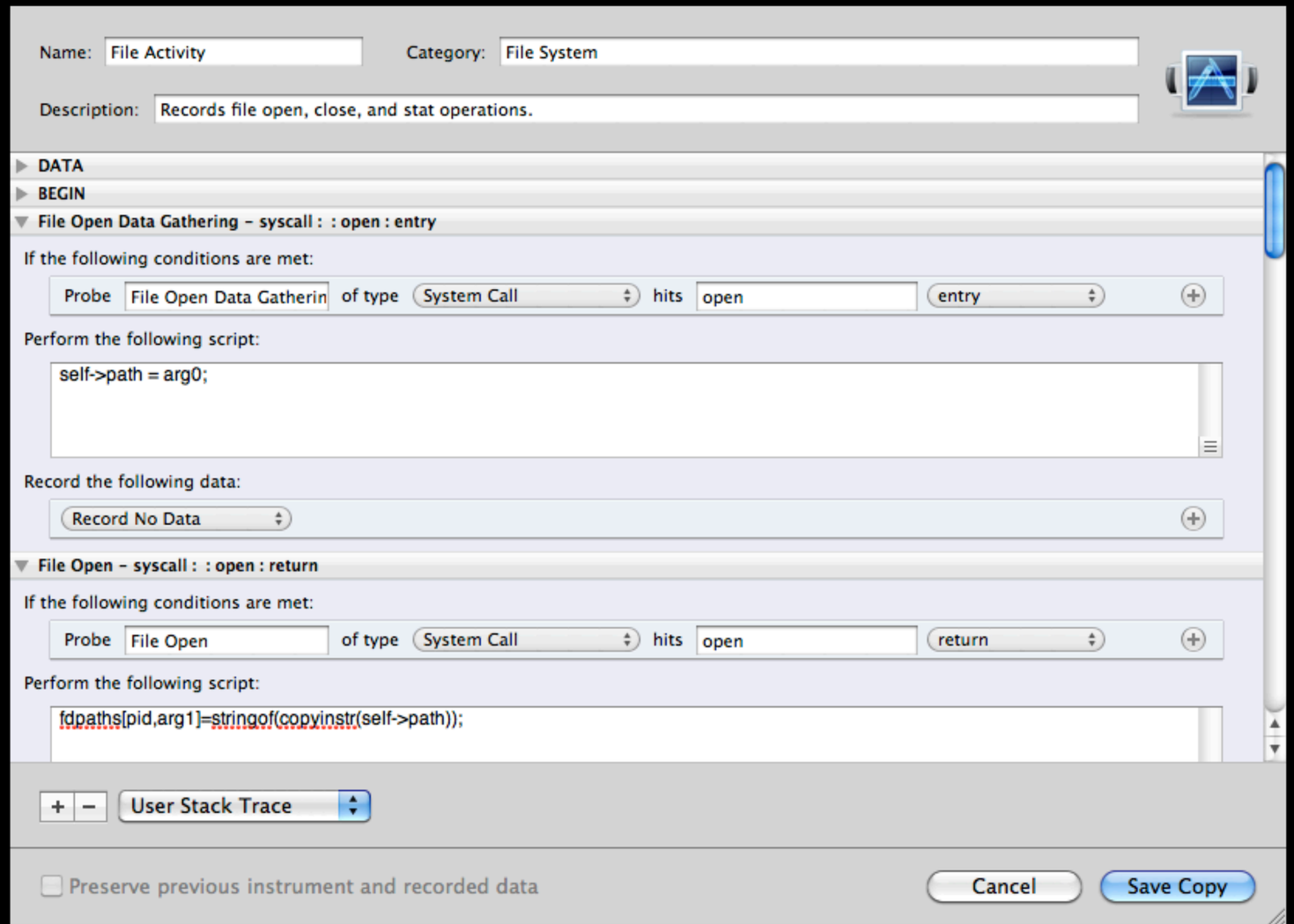
Instruments

Instruments



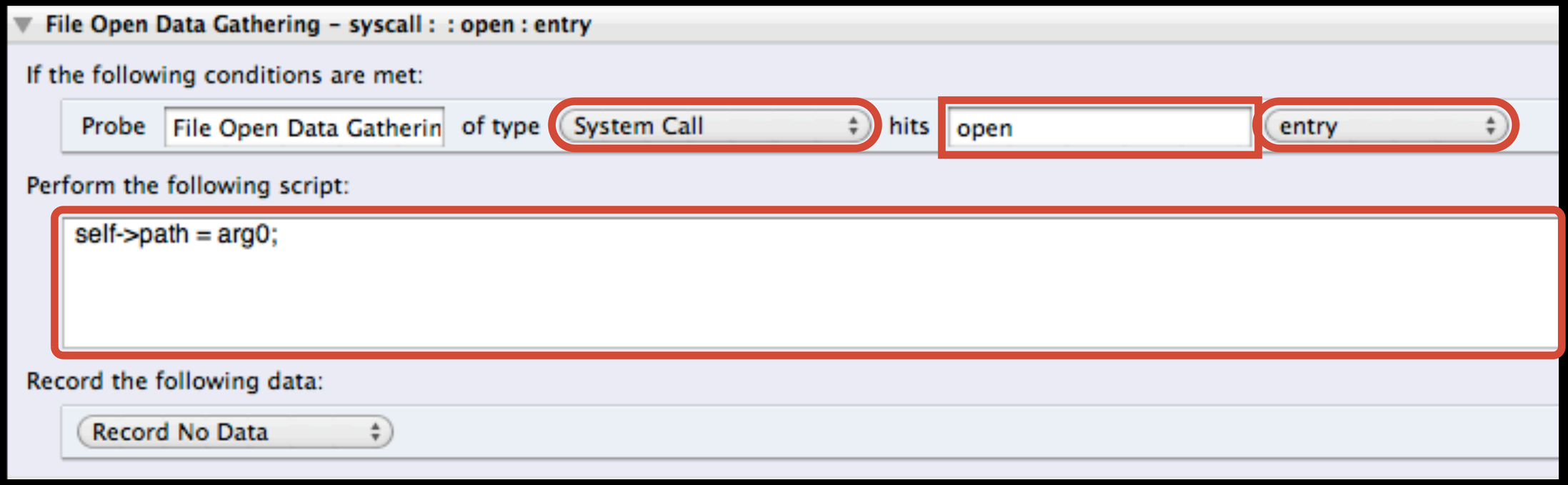
Instruments

File Activity Instrument



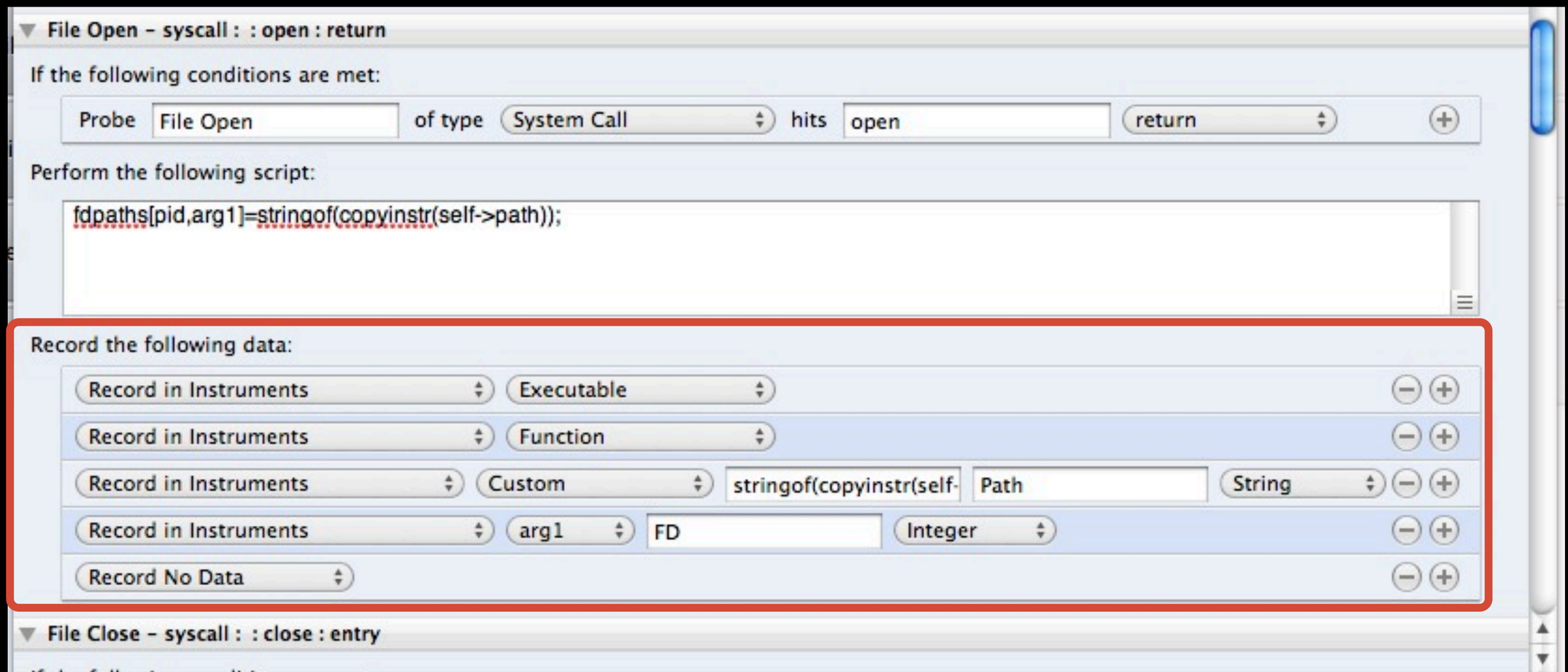
Instruments

File Activity Instrument



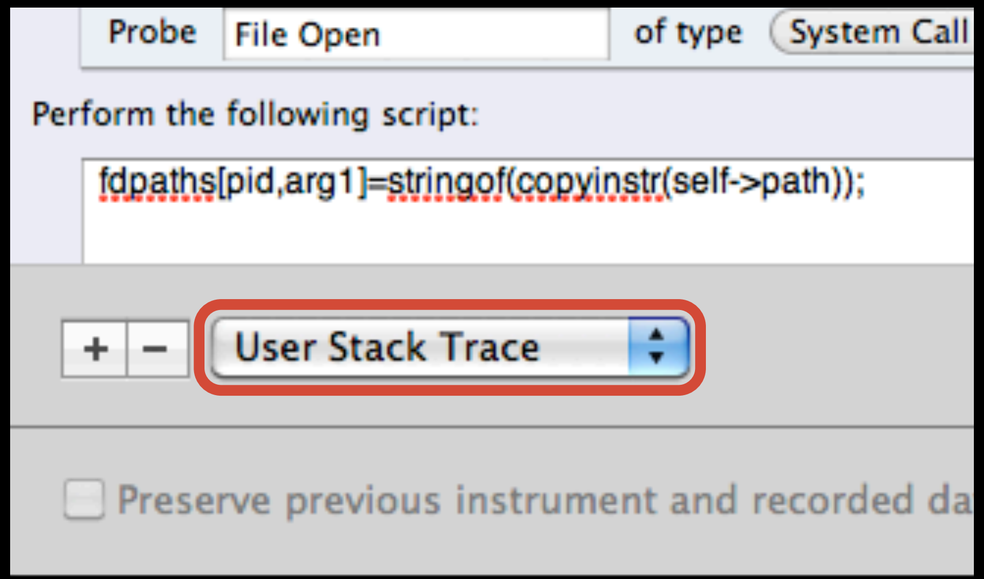
Instruments

File Activity Instrument



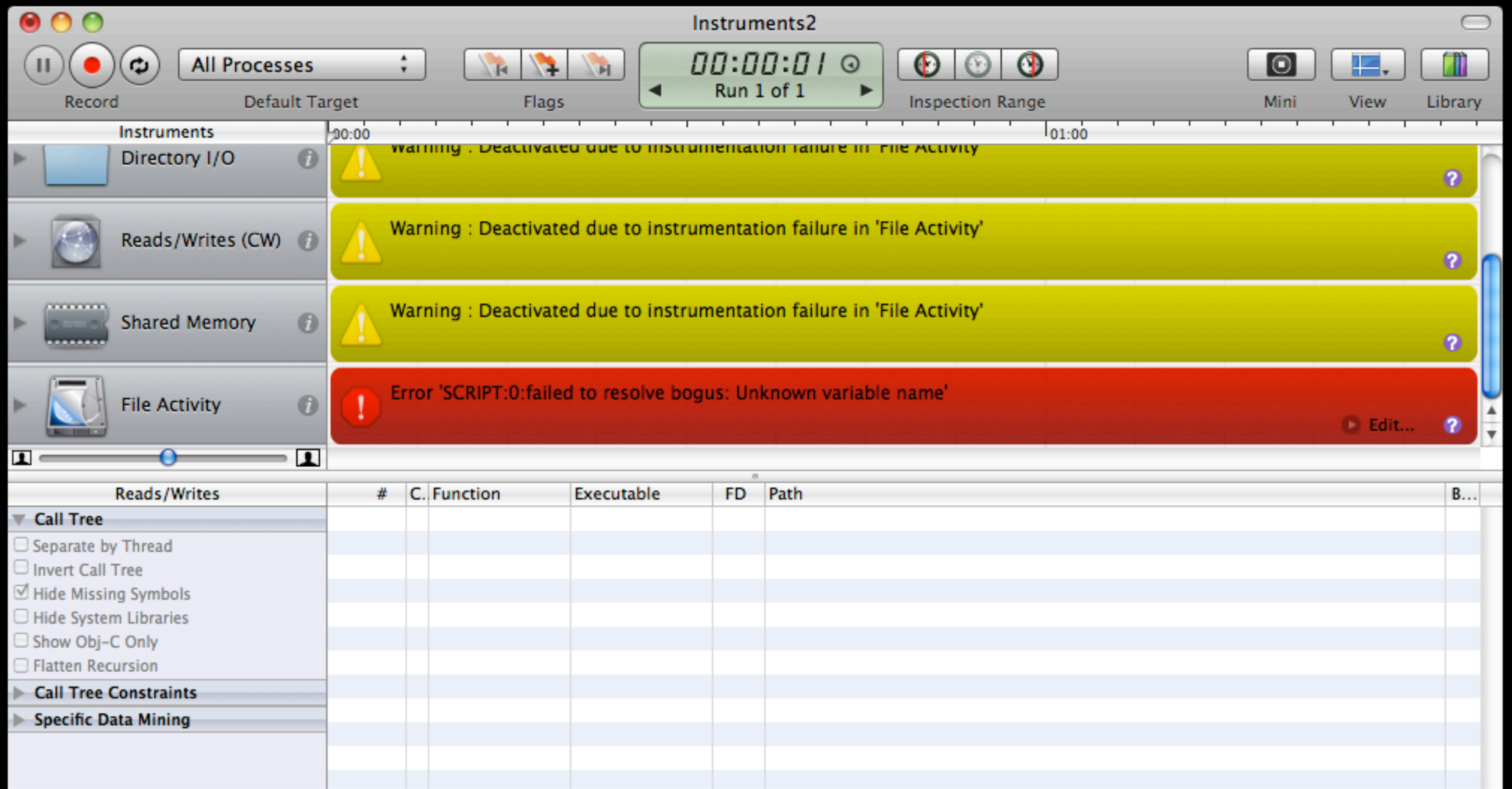
Instruments

DTrace Instrument



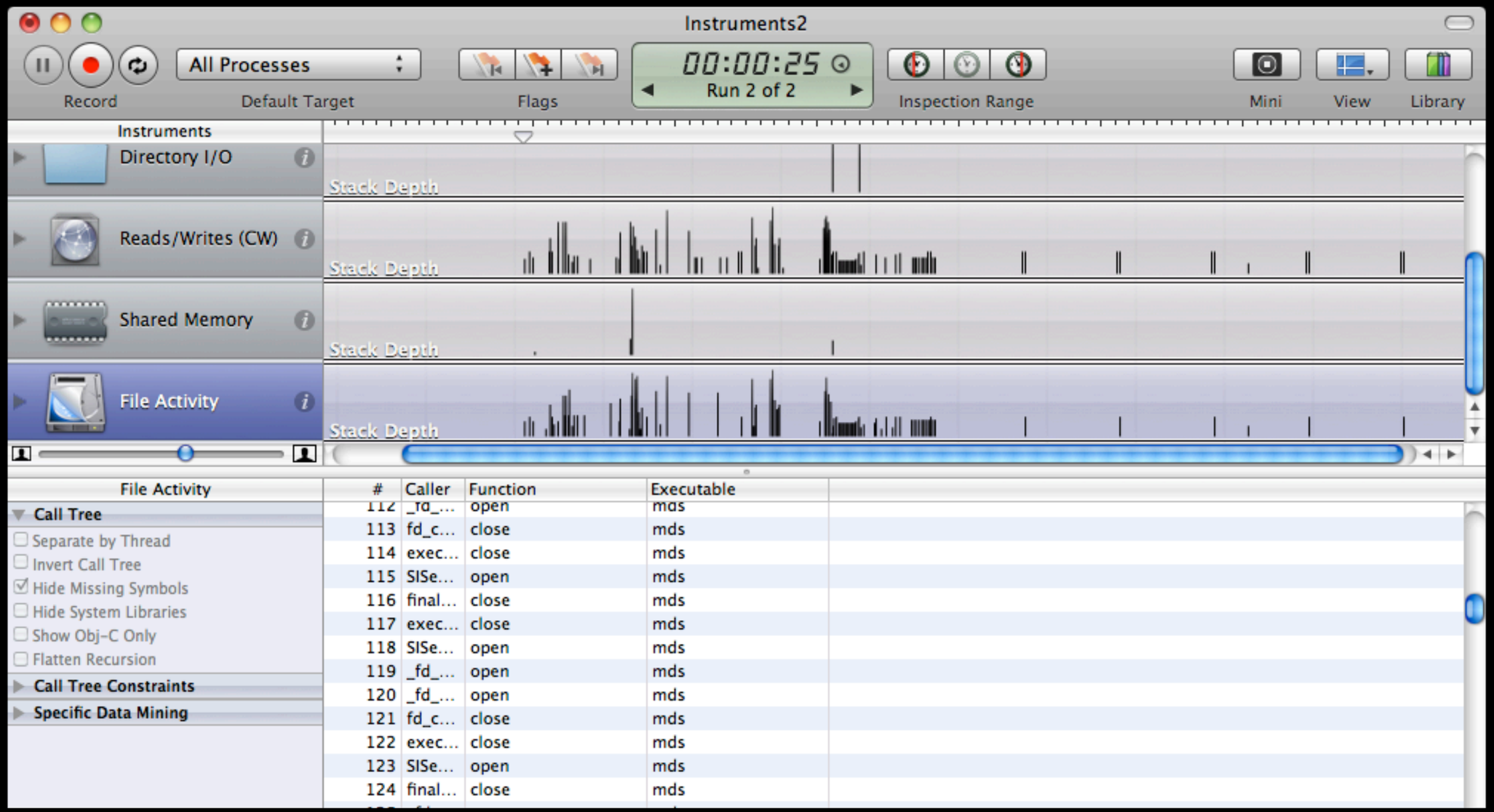
Instruments

DTrace Instrument



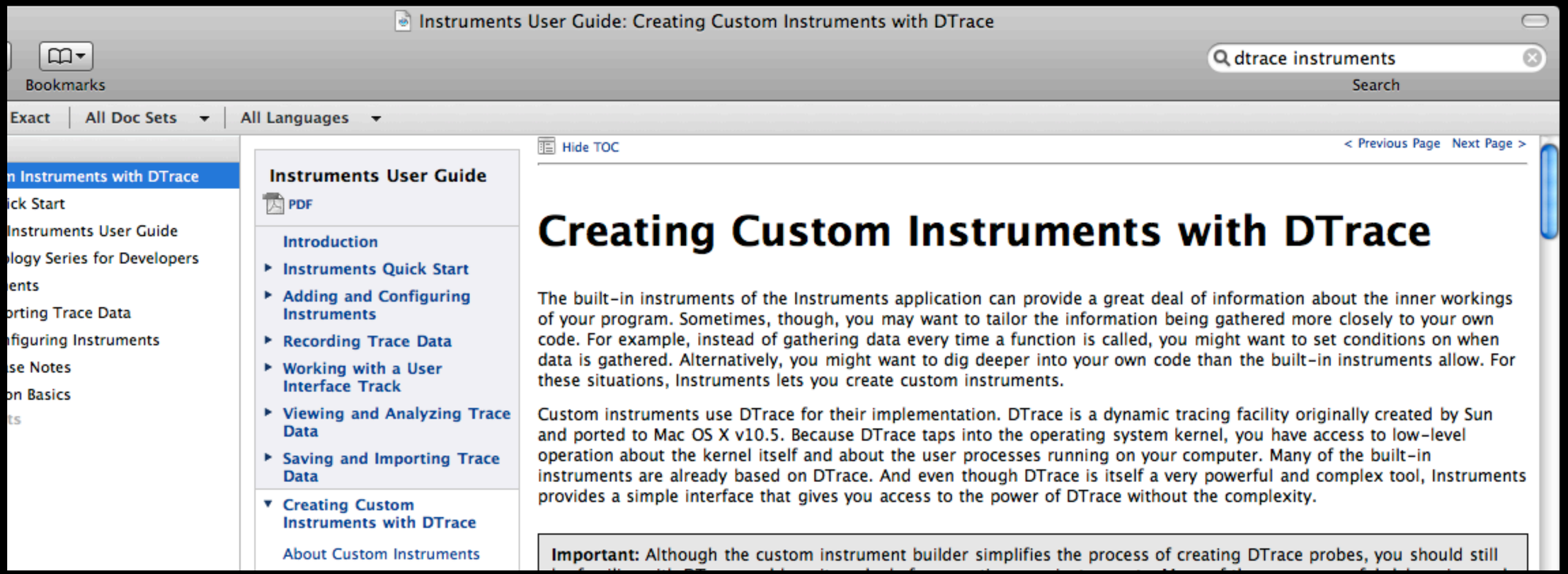
Instruments

DTrace Instrument



Custom Instruments

To find out more…



Summary

- DTrace adds tracing dynamically
  - Zero cost when not in use
- DTrace can trace, but it can also analyze
  - printf(), ustack(), and stack()
  - Aggregates, histograms, counters
- Traces more than just one app
  - Traces many apps
  - Kernel events
- Custom instruments

Documentation

- DTrace
  - http://wikis.sun.com/display/DTrace/Documentation
  - http://developers.sun.com/solaris/articles/dtrace_example.html
- DTrace Toolkit
  - http://opensolaris.org/os/community/dtrace/dtracetoolkit
  - Great tools like dtruss for tracing system calls

Advanced Debugging

Jim Ingham
Debugger Engineer

Introduction

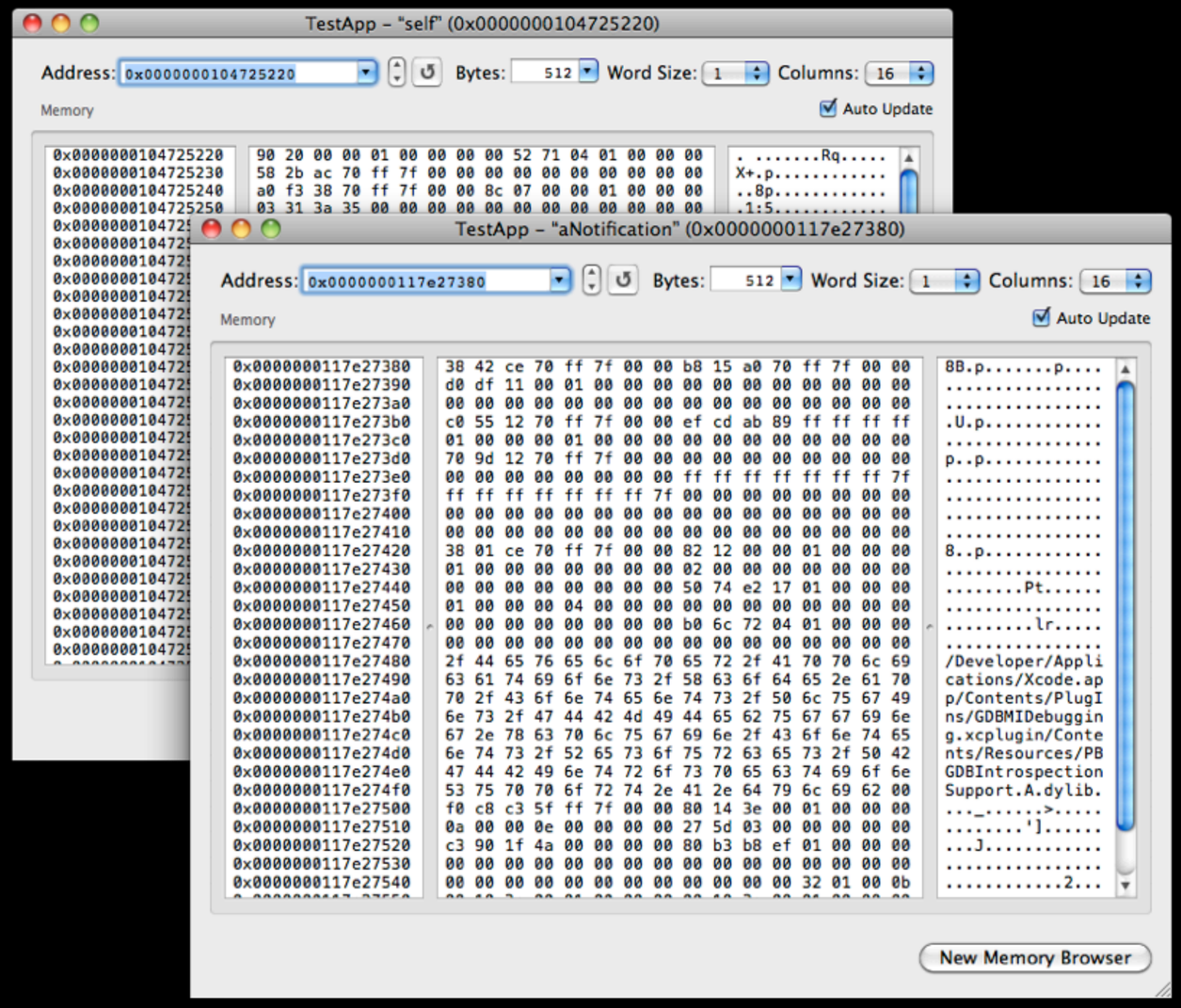
- Debugging requires a diverse bag of tricks
- We'll present a couple of flexible tools
- Also an assortment of more specific tools

What You'll Learn

- New features in the Xcode Debugger
- Data formatters for easier variable displays
- Some facilities for easier debugging of threads
- Handy gdb command language tricks

New Features in Xcode 3.2

- Memory windows



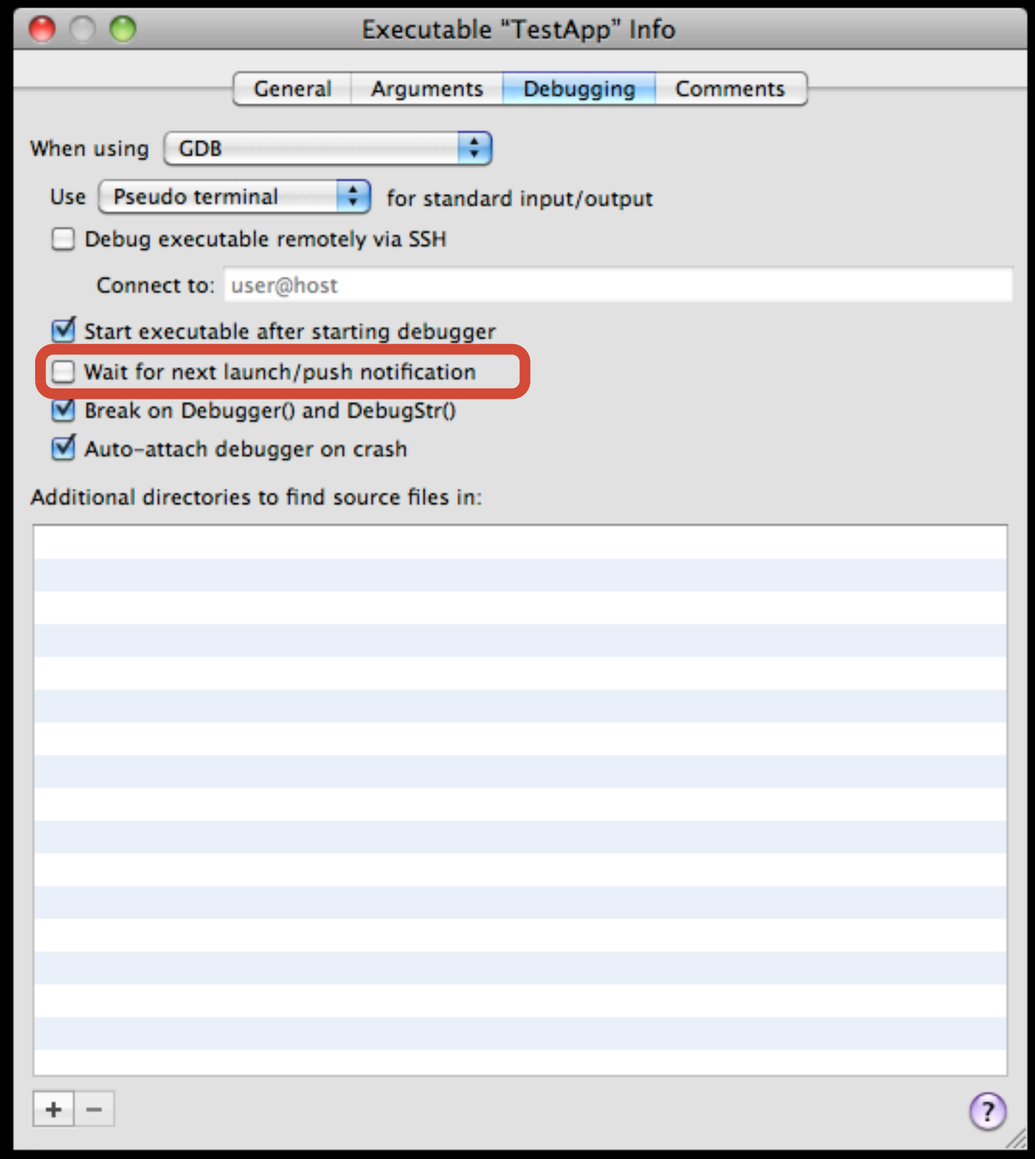
New Features in Xcode 3.2

- Memory windows
- Attaching to programs

Attaching to Programs

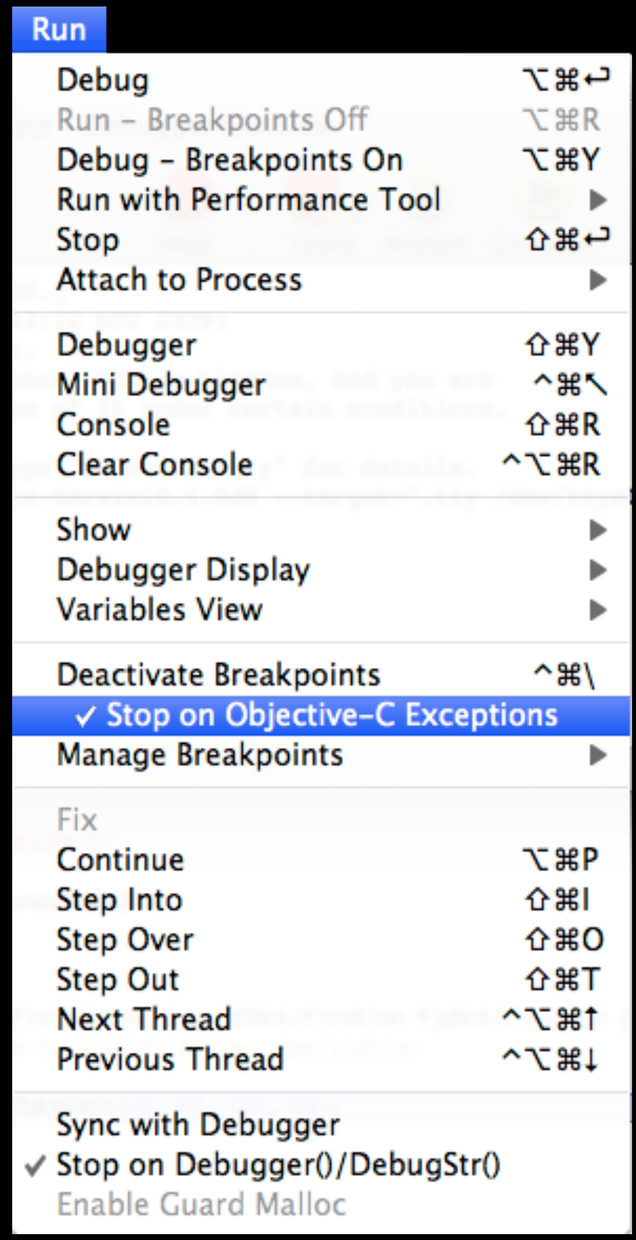
- Xcode already supports attach
- Some programs are hard to catch
  - launchd daemons
  - Worker programs
  - On the iPhone, push notifications
- In Xcode use the Executable Editor
  - Project->Edit Active Executable
- In gdb, use “attach -waitfor”

```
(gdb) attach -waitfor MyProcess
Waiting for process 'MyProcess' to launch.
```



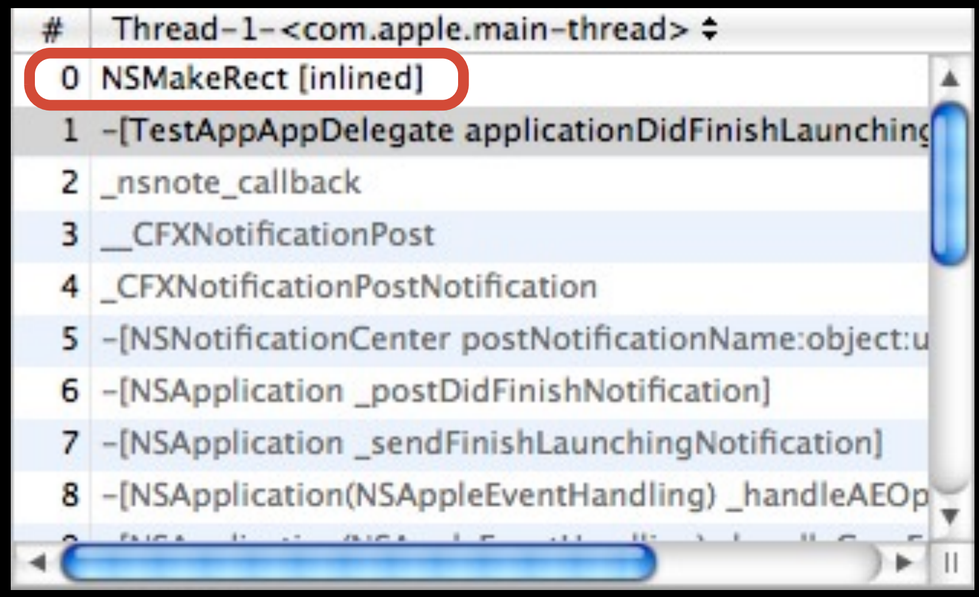
New Features in Xcode 3.2

- Memory Windows
- Attaching to programs
- Stop on Obj-C exceptions



New Features in Xcode 3.2

- Memory Windows
- Attaching to programs
- Stop on Obj-C exceptions
- Inlined code support
  - step, next, and finish work normally
  - Inlined frames show up in backtrace
  - Set breakpoints by file and line



New Features in Xcode 3.2

- Memory Windows
- Attaching to programs
- Stop on Obj-C exceptions
- Inlined code support
- Thread introspection
  - More later

New Features in Xcode 3.2

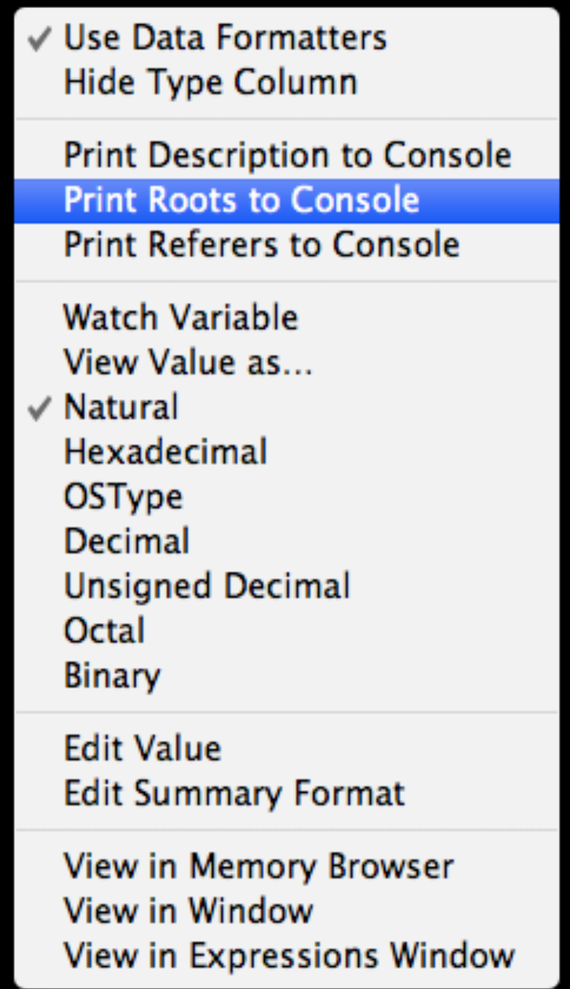
- Memory Windows
- Attaching to programs
- Stop on Obj-C exceptions
- Inlined code support
- Thread introspection
- GC Memory analysis tools

GC Memory Problems

- Find why objects are not going away when done
- Some other object has a reference to it
- From Xcode, Print Roots/Print Referers in popup
- From Console:

```
(gdb) info gc-roots <EXPRESSION>
```

```
(gdb) info gc-references <EXPRESSION>
```



New Features in Xcode 3.2

- Memory Windows
- Attaching to programs
- Stop on Obj-C exceptions
- Inlined code support
- Thread introspection
- GC Memory analysis tools
- Malloc history in the debugger

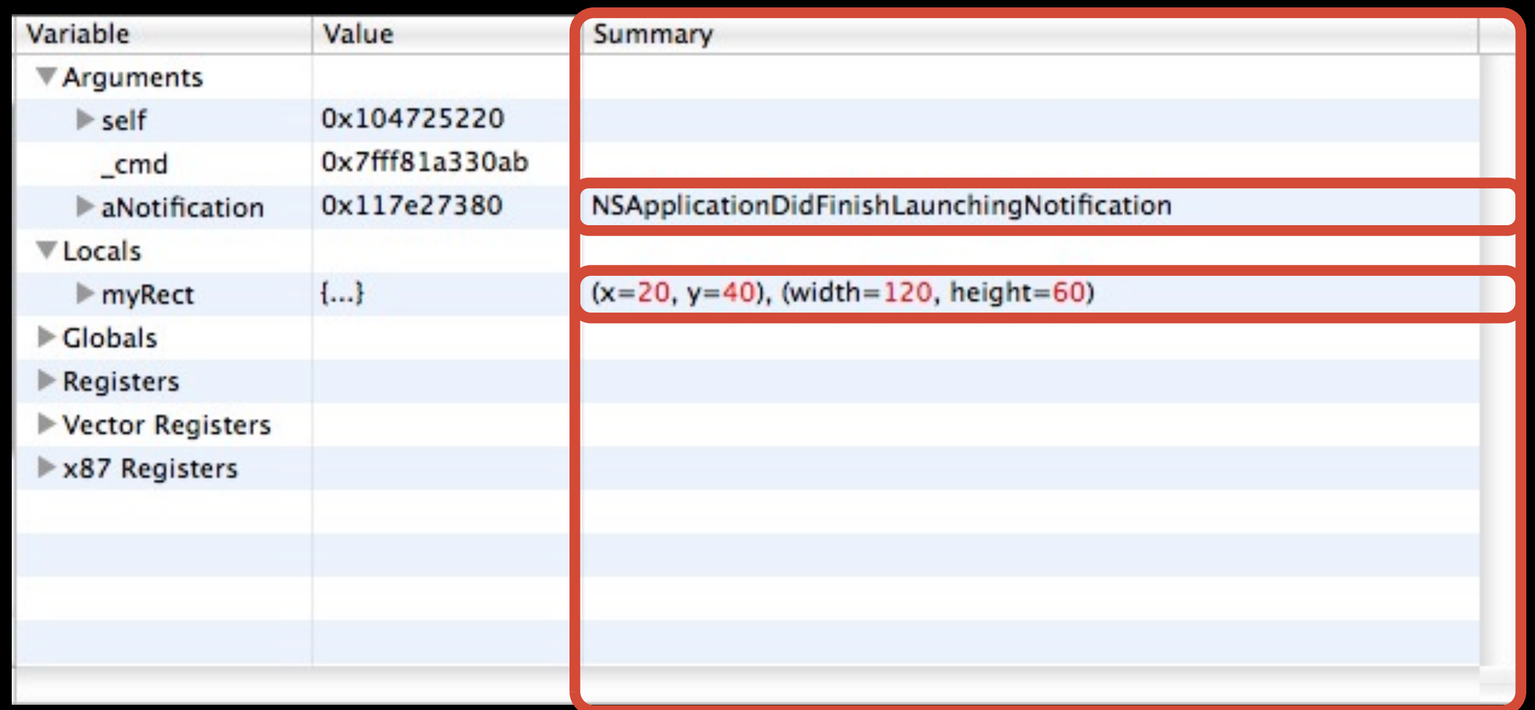
Malloc History

- Like malloc_history program but with full stack traces
- Set MallocStackLoggingNoCompact to YES in environment
- From Console:

```
(gdb) info malloc-history <EXPRESSION>
```

```
(gdb) info malloc-history -range <EXPRESSION>
```

Data Formatters



Data Formatters

- Fill Summary column of Local Variables view
- Helpful to
  - Avoid hunting around among the ivars
  - Do special formatting—e.g., casting…
  - Filter value through expression
  - Even insert custom made functions
    - We won't talk about this—details in:
      - http://developer.apple.com/samplecode/WcharDataFormatter
- Docs: Xcode Debugging Guide->Viewing Variables and Memory->Using Data Formatters

Formatter General Rules

- Results displayed in “Summary” column
- Key off variable’s type
  - typedef’s get separate formatters
- Entered in Summary column
  - Double-click for editing mode
  - Switch off row to see value update
- Stored in plist files:
  - User formats in ~/Library/Application Support/Developer/Shared/Xcode/CustomDataViews/CustomDataViews.plist

The Formatter String

- Annotation text: Can be anything
- Value subelements
  - Two kinds:
    - Structure references—% delimiter
    - General expressions—{} delimiter
  - Value origin modifiers
    - The subelement has a type and a value
    - :v - From subelement's Value column—default
    - :s - From subelement's Summary column
    - :t - The dynamic type of the subelement
    - Modifier appended to subelement

```
Struct: %origin.x%:v Evil: {$VAR++}:v
```

Demo

Greg Clayton
Debugger Engineer

Threading Tricks

- Identify your threads in debugger
- Get more thread-specific information
- Run just one thread to make debugging simpler

Identifying Threads

- pthread name (Snow Leopard only):

```
pthread_setname_np (“MyApp:my_first_thread”);
```

  - Choose some unique name
  - -[NSThread setName:] doesn’t set this
- GCD queue label:

```
dispatch_queue_create (“com.myCompany.myQueue”, attrs);
```

- Use reverse DNS form for queue names
- Thread and queue names appear in Debugger thread popup
  - ⌘⌥^-↑ and ⌘⌥^-↓ to switch threads

Inspecting Threads in Console

- List threads with “info threads” command

```
(gdb) info threads
* 1  port# 0x903  "MyApp:my_first_thread"  main () at main.c:8
```

gdb’s thread id

Mach thread port

Thread name

Current function or pc

Inspecting Threads in Console

- Detailed information with “info thread <NUM>”

```
(gdb) info thread 1
Thread 1 has current state "WAITING"
Mach port #0x903 (gdb port #0x4603)
frame 0: foo () at foo.c:5
pthread ID: 0x7fff70137e60
system-wide unique thread id: 0xe97d
dispatch queue name: "com.apple.main-thread"
dispatch queue flags: 0x0
total user time: 0x3d2c28
total system time: 0x641518
scaled cpu usage percentage: 0
scheduling policy in effect: 0x1
run state: 0x3 (WAITING)
flags: 0x0
number of seconds that thread has slept: 0
current priority: 31
max priority: 63
suspend count: 0.
```

Dispatch Queue Name

Inspecting Threads in Console

- Operate on multiple threads with “thread apply”

```
(gdb) thread apply all backtrace 5

Thread 8 (process 503):
#0  0x00007fff80b94f46 in __workq_kernreturn ()
#1  0x00007fff80b95429 in _pthread_wqthread ()
#2  0x00007fff80b95059 in start_wqthread ()

Thread 7 (process 503):
#0  0x00007fff80b9f07a in __semwait_signal ()
#1  0x00007fff80bdfb1a in nanosleep ()
#2  0x00007fff80beacf4 in usleep ()
#3  0x00007fff85ca8c91 in -[NSUIHeartBeat _heartBeatThread:] ()
#4  0x00007fff86f937e0 in __NSThread__main__ ()
(More stack frames follow...)


...
```

Hand-Scheduling Threads

- Running only one thread causes deadlocks
- Debugger lets all threads run:
  - When you step over a function
  - When you “continue”
- Can break in another thread
- This can make debugging very confusing
- Can you stop other threads and just watch one?

scheduler-locking Command

- “scheduler-locking” command does just this

```
(gdb) set scheduler-locking on
(gdb) set scheduler-locking off
```

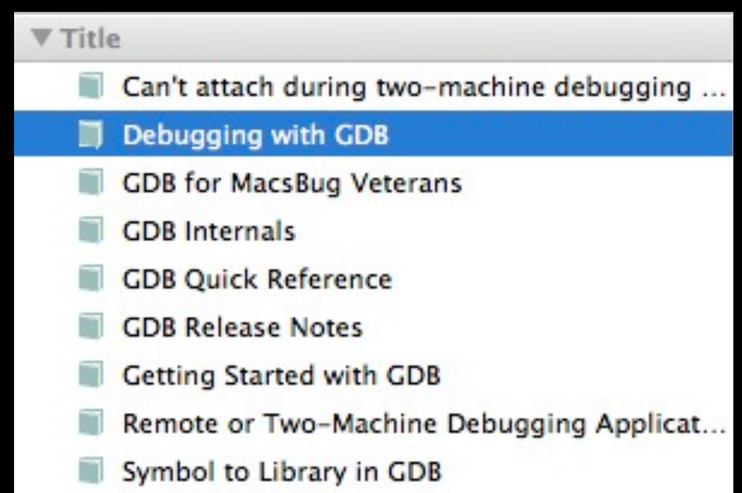
- “on”: Only the current thread will run
  - Thread popup changes “current thread”
- “off”: All threads run
- No UI in Xcode for this

Provisos…

- Running one thread may stall that thread
  - e.g., blocking on lock held by another thread
- Symptom: Your program stops doing anything…
- Just interrupt (^C or Pause) and
  - Turn scheduler locking “off” and continue
  - Find the thread holding lock, run to release it

Command Language Tricks

- The gdb command language has powerful features
  - "help" and "apropos" commands
- "printf": Format readable output
- Convenience variables ($foo)
- Logic constructs
- "define": Can command sequences for reuse
- Docs
  - Open Xcode Help
  - Search for GDB
  - Select "Debugging with GDB"



printf Command

- Same format as C's printf
- Done in gdb, not in your app

```
(gdb) printf “An int: %d\nA string: %s\n”, my_int, my_string
An int: 50
A string: Some string.
```

Convenience Variables

- Begin with “$”
- Define with the “set” command
  - Value and type come from expression
- Work the same as the program’s variables

```
(gdb) set $tmp = (long long) my_int
(gdb) ptype $tmp
type = long long
(gdb) printf "Value is: %lld\n", $tmp
Value is: 50
```

Automatic Convenience Variables

- Registers
  - Machine: $rax, $rbx, etc.
  - Synthetic: $pc, $sp, $fp
- The “print” command makes a variable for result
  - Called $<NUM>

```
(gdb) print my_struct
$1 = (struct struct_2 *) 0x7fff5fbff170
(gdb) print $1->ptr
$2 = (struct struct_1 *) 0x7fff5fbff180
(gdb) print *$
$3 = {
  some_value = 10,
  other_value = 20
}
```

Use “$1” in place of original expression

“$” refers to the last $<NUM> variable

define Command

- Defines procedure for reuse
- Arguments are passed as $arg0, $arg1, etc.
  - Number of args in $argc
- “document” to document your commands
- “help user” lists user commands
  - Also shows the first line of your documentation
- “help commandName” shows full documentation
- Put in ~/.gdbinit —loaded on startup
- Or use “source” command

Example: Traverse Linked List

```
struct my_list {
  int value;
  char *name;
  struct my_list *next;
} *my_list_head;
```

- Task—dump elements matching an input value
  - Make it into a user command
  - First arg is the list head
  - Second arg is the match value

Filter Linked List as Command

```
define look_for_value
  set $elem = $arg0
  set $test_val = $arg1
  while $elem != 0
    if $elem->value == $test_val
      printf “Name: \”%s\” value: %d\n”,$elem->name, $elem->value
      print $elem
    end
    set $elem = $elem->next
  end
end
```

Arguments

```
define look_for_value
  set $elem = $arg0
  set $test_val = $arg1
  while $elem != 0
    if $elem->value == $test_val
      printf “Name: \”%s\” value: %d\n”,$elem->name, $elem->value
      print $elem
    end
    set $elem = $elem->next
  end
end
```

while Command

```
define look_for_value
  set $elem = $arg0
  set $test_val = $arg1
  while $elem != 0
    if $elem->value == $test_val
      printf “Name: \”%s\” value: %d\n”,$elem->name, $elem->value
      print $elem
    end
    set $elem = $elem->next
  end
end
```

if Command

```
define look_for_value
  set $elem = $arg0
  set $test_val = $arg1
  while $elem != 0
    if $elem->value == $test_val
      printf “Name: \”%s\” value: %d\n”,$elem->name, $elem->value
      print $elem
    end
    set $elem = $elem->next
  end
end
```

print Command

```
define look_for_value
  set $elem = $arg0
  set $test_val = $arg1
  while $elem != 0
    if $elem->value == $test_val
      printf “Name: \”%s\” value: %d\n”,$elem->name, $elem->value
      print $elem
    end
    set $elem = $elem->next
  end
end
```

- Print the element—you can get back to it later

Document It

```
define look_for_value
  set $elem = $arg0
  set $test_val = $arg1
  while $elem != 0
    if $elem->value == $test_val
      printf “Name: \”%s\” value: %d\n”,$elem->name, $elem->value
      print $elem
    end
    set $elem = $elem->next
  end
end
document look_for_value
  Searches for values in the linked list my_linked_list
  The first argument is the head of the linked list
  Prints all elements whose VALUE matches second argument
end
```

Run It

```
(gdb) help look_for_value
Searches for values in the linked list my_linked_list
The first argument is the head of the linked list
Prints all elements whose VALUE matches second argument
(gdb)
```

Run It

```
(gdb) help look_for_value
Searches for values in the linked list my_linked_list
The first argument is the head of the linked list
Prints all elements whose VALUE matches second argument
(gdb) look_for_value my_list_head 3
Name: "Item 98" value: 3
$21 = (struct my_list *) 0x100103dc0
Name: "Item 93" value: 3
$22 = (struct my_list *) 0x100103aa0
Name: "Item 88" value: 3
...
(gdb)
```

Run It

```
(gdb) help look_for_value
Searches for values in the linked list my_linked_list
The first argument is the head of the linked list
Prints all elements whose VALUE matches second argument
(gdb) look_for_value my_list_head 3
Name: "Item 98" value: 3
$21 = (struct my_list *) 0x100103dc0
Name: "Item 93" value: 3
$22 = (struct my_list *) 0x100103aa0
Name: "Item 88" value: 3
...
(gdb) print *$22
$41 = {
  value = 3,
  name = 0x100103ac0 "Item 93",
  next = 0x100103a00
}
(gdb)
```

Summary

- We saw some new Xcode features
- Learned how to use data formatters
- Some helpful threading tricks
- And got a brief intro to the gdb command language

More Information

Michael Jurewitz
Developer Tools and Performance Evangelist
jurewitz@apple.com

Documentation

Debugging with GDB
Xcode Help

Related Sessions

What's New in Instruments	Pacific Heights Wednesday 4:00PM
iPhone Performance Optimization with Instruments	Marina Thursday 5:00PM

Labs

Advanced Debugging and Performance Analysis	OS X Lab B Thursday 9:00AM

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09